LLM-VeriPPA: Power, Performance, and Area-aware Verilog Code Generation and Refinement with Large Language Models

Anonymous ACL submission

Abstract

As Large Language Models (LLMs) gain increasing prominence across a variety of domains and inspired by their remarkable ability to generate high-quality content in response to human language instructions. This study delves into the application of LLMs within the field of hardware design, specifically in the generation and refinement of Verilog code. We introduce a novel framework VeriPPA designed to assess and enhance LLM efficiency in this specialized area. Our method includes generating initial Verilog code using LLMs, followed by a unique two-stage refinement process. The first stage focuses on improving the functional and syntactic integrity of the code, while the second stage aims to optimize the code in line with Power-Performance-Area (PPA) constraints, an essential aspect of effective hardware design. This dual-phase approach of error correction and PPA optimization has led to notable improvements in the quality of LLM-generated Verilog code. Our framework achieves a success rate of 81.37% for syntactic correctness and 62.0% for functional accuracy in code generation, surpassing current state-of-the-art (SOTA) methods, e.g., 73% for syntactic correctness and 46% for functional accuracy. These results highlight the potential of LLMs in handling complex technical areas, and indicate an encouraging development in the automation of hardware design processes. Our source codes can be found on Github[1].

1 Introduction

With Moore's law driving increased design complexity and chip capacity, the chip design requires more effort. Machine learning (ML) has successfully integrated into chip design for logic synthesis (Haaswijk et al., 2018; Hosny et al., 2020), placement (Ward et al.), routing (Liang et al., 2020; Maarouf et al., 2018), testing (Chen et al., 2012; Wang et al., 2018; Liu et al., 2019), and verification (Fine and Ziv, 2003; Hu et al., 2018). The popularity of agile hardware design exploration has been on the rise due to the growth of large language models (LLMs). A promising direction is using natural language instruction to generate hardware description language (HDL), e.g., Verilog, aiming to greatly lower hardware design barriers and increase design productivity, especially for users who do not possess extensive expertise in chip design. Despite the efforts, Verilog benchmarking has unique challenges in terms of the wide range of hardware designs (Liu et al., 2023).

Two orthogonal research trends have both attracted enormous interests (Thakur et al., 2023; Lu et al., 2023; Liu et al., 2023; Blocklove et al., 2023; Chang et al., 2023). The *first trend* is efficiently finetuning LLMs such as CodeGen (Nijkamp et al., 2022), with representatives works such as Thakur *et al.* (Thakur et al., 2023), Chip-Chat (Blocklove et al., 2023), Chip-GPT (Chang et al., 2023). However, due to limited Verilog data sources, these works mainly target the scale of simple and small circuits (e.g., <20 designs with a medium of <45 HDL lines) (Lu et al., 2023). The relatively low scalability and solution quality have propelled the *second trend* – enrich Verilog source. Like oil, data is an immensely valuable resource. One could not generate high quality HDL codes without having LLMs trained on vast amount of such data. RTLLM (Lu et al., 2023) and VerilogEval (Liu et al., 2023) introduce specialized benchmarking framework (i.e., 30 designs from RTLLm and 156 designs from HDLBits (HDLBits, 2023) from VerilogEval) to assess the generation quality of LLMs. However, they either do not offer Power, Performance, and Area (PPA) analysis for the generated codes (e.g., VerilogEval), or the generated Verilog codes are directly extracted and synthesized using commercial tools to obtain PPA results, without considering PPA feedback (e.g., RTLLM). Thus,

[1] https://anonymous.4open.science/r/LLM-VeriPPA-10B7

their solution quality is still limited.

In this work, as the first attempt, we integrate power, performance, and area-constraints into Verilog generation, and propose VeriPPA, a open-source framework with multi-round Verilog generation and error feedback, shown in Figure 1. We first generate initial Verilog codes using LLMs, followed by a unique two-stage refinement process. The first stage focuses on improving the syntax and functionality, while the second stage aims to optimize the code in line with PPA constraints, an essential aspect to ensure hardware design quality. Compared with state-of-the-arts (SOTAs), e.g., RTLLM (Lu et al., 2023), VerilogEval (Liu et al., 2023), our VeriPPA achieves a success rate of 62.0% (+16%) for functional accuracy and 81.37% (+8.3%) for syntactic correctness in Verilog code generation. Our key contributions are summarized here:

- We use the detailed error diagnostics from the iverilog simulator (Williams, 2023), and pinpoint the exact location of syntactic or functional discrepancies as indicated by testbench failures as new prompts. We use multi-round generation to enhance the syntax and functionality correctness.
- To further ensure that the generated Verilog codes are *synthesizable*, and design quality (i.e., PPA) is sound, we use Synopsys Design Compiler to perform logic synthesis (and technology mapping) on the open source ASAP 7nm Predictive PDK, and check all designs' warnings/errors, and PPA report. We then integrate these PPA reports and warnings/errors with our PPA goal into the next round prompt for further refinement.
- We incorporate in-context learning (ICL) to significantly improve the LLM performance in generating Verilog codes with only a few demonstration examples, especially when labeled data are scarce. By carefully selecting diverse text-to-Verilog pairs, ICL demonstrates superior performance and generalization capabilities compared to fine-tuning in limited example scenarios, thus increasing the performance of Verilog code generation.

2 Background and Related works

Finetune LLMs. Thakur *et al.* (Thakur et al., 2023) advocate for the fine-tuning of open-source LLMs such as CodeGen (Nijkamp et al., 2022) to specifically generate Verilog code tailored for target designs. Subsequently, Chip-Chat (Blocklove et al., 2023) delves into the intricacies of hardware design using LLMs, highlighting the markedly superior performance of ChatGPT compared to other open-source LLMs. Chip-GPT (Chang et al., 2023) also focuses on the task of register-transfer level (RTL) design by leveraging the capabilities of ChatGPT. These studies pave the way for a promising future where language models play a pivotal role in facilitating and enhancing various aspects of agile hardware design exploration. However, these works mainly target the scale of simple and small circuits (e.g., <20 designs with a medium of <45 HDL lines), as pointed out in (Lu et al., 2023).

Enrich Verilog Source. Several recent efforts focuse on enriching Verilog codes. RTLLM (Lu et al., 2023) introduces a benchmarking framework consisting of 30 designs that are specifically aimed at enhancing the scalability of benchmark designs. Furthermore, it utilizes effective prompt engineering techniques to improve the generation quality. VerilogEval (Liu et al., 2023) assesses the performance of LLM in the realm of Verilog code generation for hardware design and verification. It comprises 156 problems from the Verilog instructional website HDLBits. However, VerilogEval (Liu et al., 2023) does not offer PPA analysis for the generated codes. In RTLLM, the generated Verilog codes are directly extracted and synthesized using commercial tools to obtain PPA results, without PPA constraint-based feedback. Thus they suffer from limited generation quality.

3 Framework

3.1 Design Overview

In our VeriPPA framework, as illustrated in Figure 1, we use a text-based description (.txt file) of Hardware Design, designated as L, to serve as input/prompt for the LLMs. L details the module name, and specifies both input and output signals with the corresponding bit widths. We use LLM to parse L and subsequently generate the corresponding Verilog codes, V. V is then subjected to a rigorous validation sequence, beginning with a Simulator that checks both syntax and functionality. In the first loop, if unsuccessful, we will input the outcomes along with any syntax and functionality errors, into the LLM for the generation of subsequent attempts. If successful, the code undergoes

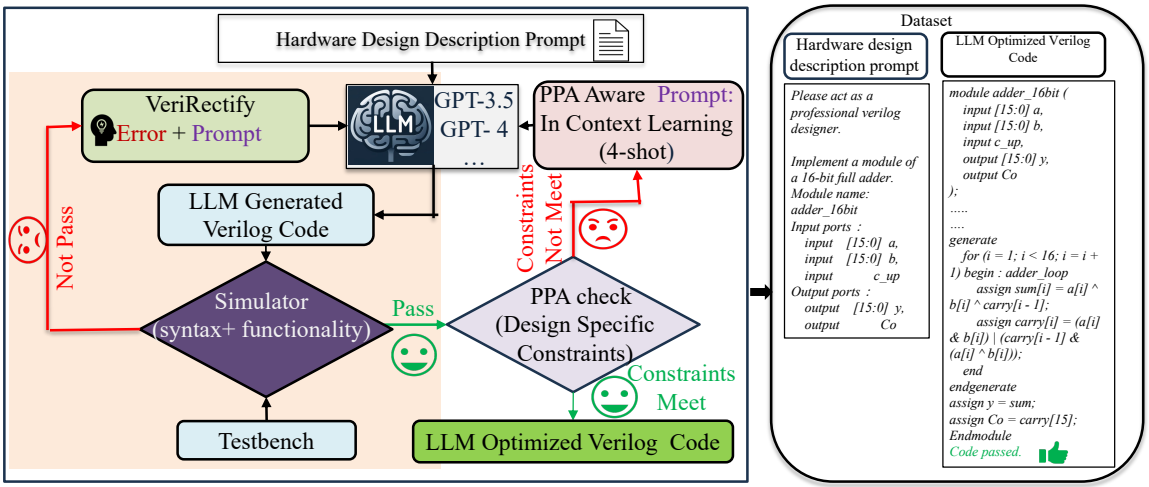

Figure 1: This visualization captures the step-by-step process where an LLM synthesizes Verilog codes from hardware design prompts, with the ensuing code subjected to thorough validation by a Simulator and scrutinized for adherence to Power-Performance-Area (PPA) checks.

Power-Performance-Area (PPA) checks to ensure compliance with constraints. In the second loop, we check all designs' warnings/errors during logic synthesis and using PPA reports. If not satisfied, both the design and its corresponding PPA report will be fed back to the VeriRectify (Section 3.3) for refinement. This validation workflow ensures that the LLM-generated Verilog codes not only meet functional specifications but also is optimized for PPA considerations.

3.2 Code Generation and Testing

VeriPPA incorporates the ICARUS Verilog simulator (Williams, 2023) to automate the evaluation (testing) of the generated codes. In contrast to high-level program languages such as Python, Verilog requires the use of testbenches, $T = \{T_1, T_2, \ldots, T_m\}$, to systematically assess the code's functionality, encompassing a wide array of test scenarios. Integrating the ICARUS Verilog simulator into VeriPPA provides immediate feedback on the code's syntactical and operational integrity. The ICARUS Verilog simulator could pinpoint the exact location of syntactic errors or functional fails based on testbench test case failures. This integrated approach contrasts with frameworks such as RTLLM (Lu et al., 2023), where an external simulator is used to check the correctness of the generated Verilog codes.

3.3 VeriRectify

The refinement process, termed "*VeriRectify*," is pivotal to ensuring the correctness of the generated Verilog codes. In Figure 2, the top box displays the syntax errors and functional fails found in the

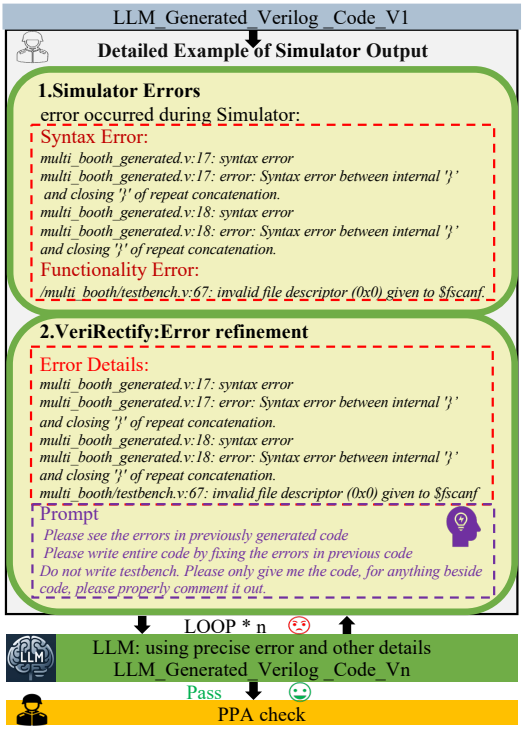

Figure 2: The diagram illustrates the process of syntactic and functional code verification.

output of simulation (e.g., booth_multiplier) from the ICARUS Verilog simulator. During the rectification phase, we utilize a tailored prompt (Integrating the error details with additional text) for the LLMs, for error correction. This approach allows the LLM to specifically address the encountered issues, which is fundamentally different from RTLLM (Lu et al., 2023) (using a generalized prompt for all designs). Figure 2 shows that the LLM effectively amends the error in the Verilog codes for the booth_multiplier design.

Algorithm 1: Multi-Round Verilog Code Generation using LLM

Require: User prompt P, iteration limit K
Ensure: Correct Verilog code V_{final} or V_K
$i \leftarrow 0$
$conv \leftarrow$ [] # initialize a new conversation history list
$conv \leftarrow$ Append user prompt (P) to conversation
$V_i \leftarrow$ LLM-generated code using prompt Pin history$conv$
$E_i \leftarrow$ Error detection function using simulator$(D(V_i))$
while $E_i \neq \emptyset$ AND $i < K$ **do**
  $conv \leftarrow$ Append error information (E_i) to conversation
  $V_{i+1} \leftarrow$ LLM-generated code using prompt P_{new}
  $E_{i+1} \leftarrow$ Error detection using simulator $(D(V_{i+1}))$
  $i \leftarrow i + 1$
end while
if $E_i = \emptyset$ **then**
  $V_{final} \leftarrow V_i$
else
  $V_{final} \leftarrow V_K$
end if
return V_{final}

3.3.1 Multi-round Conversation with Error Feedback

We further integrate a multi-iteration dialogue with an error feedback mechanism, analogous to human problem-solving techniques. This method is designed as a recursive function that improves the output by carefully analyzing and correcting the errors found in previous iterations. Let V_i denote the Verilog code resultant from the i^{th} iteration, and E_i represent the associated set of identified errors at this stage. Initially, V_0 is the first generated code accompanied by its detected errors E_0. Then the refinement function, $R(V_i, E_i)$, which takes as input V_i and E_i, and yields an enhanced code version V_{i+1} as output. Simultaneously, an error detection function $D(V_i)$ is employed to identify errors within V_i, generating E_i. The iterative process can be viewed as follows:

$$V_{i+1} = R(V_i, E_i) \quad \text{and} \quad E_{i+1} = D(V_{i+1}) \quad (1)$$

This process repeats until either no errors are detected or a predefined iteration limit, K is reached, shown in Algorithm 1, i.e., the iteration halts if, $D(V_{i+1})$=$\emptyset$ or $i = K$. K is empirically adjustable (say 4) based on observed diminishing returns in LLM performance improvements across iterations.

3.4 Power Performance & Area (PPA) Checking

The *VeriRectify* process ensures the design to pass both register-transfer level (RTL) syntax check and cycle-accurate functional simulation. However, RTL simulation does not guarantee that the design (Verilog code) is *synthesizable*. Furthermore, the quality of the hardware design must be measured by its power, performance, and area metrics.

Our approach takes a step further by inspecting PPA of the design V which passes the *VeriRectify* process as the following:

$$V = \begin{cases} V & \text{if PPA}(V) \text{ satisfies,} \\ \text{VeriRectify}(V, \text{PPA}(V)) & \text{otherwise.} \end{cases} \quad (2)$$

In this work, our PPA check calls Synopsys Design Compiler to perform logic synthesis (and technology mapping) on the open-source ASAP 7nm Predictive PDK (Vashishtha et al., 2017). We check all designs' warning/error messages during the logic synthesis, and the power (μW), area (μm^2), and clock (ps) for quality. When the Verilog design can be synthesized and meets the PPA goal, it results in a pass. Otherwise, both the design and its corresponding PPA report will be fed back to the VeriRectify (Section 3.3) for refinement.

3.5 In-Context Learning

LLMs have demonstrated remarkable in-context learning (ICL) capabilities (Radford et al., 2019; Brown et al., 2020). We will further improve the code quality using ICL. We use tailored prompts (Questions) with optimization strategies such as Pipelining, Clock Gating, Parallel Operation, and Hierarchical Design, associated with Verilog examples (Answers). The LLM generates the optimized Verilog codes (Answer) based on the tailored prompts (Questions).

$$q(t|v) = \prod_{k=1}^{K} q(t_k|t_{<k}), \quad (3)$$

$t_{<k}$ are the questions and answers of strategies (i.e., pielining, gate clocking). t_k is the Verilog code not passing the PPA goal. v indicates demonstration example tokens. q is parameterized by the LLM. The equation describes the in-context learning of a large language model, where the model learns to predict the next token in the sequence by considering the previous demonstration examples. In our case, we carefully select the text-to-Verilog pairs to ensure that the examples cover a range of different Verilog designs, such as addition, multiplication, single-stage design, and pipelined design.

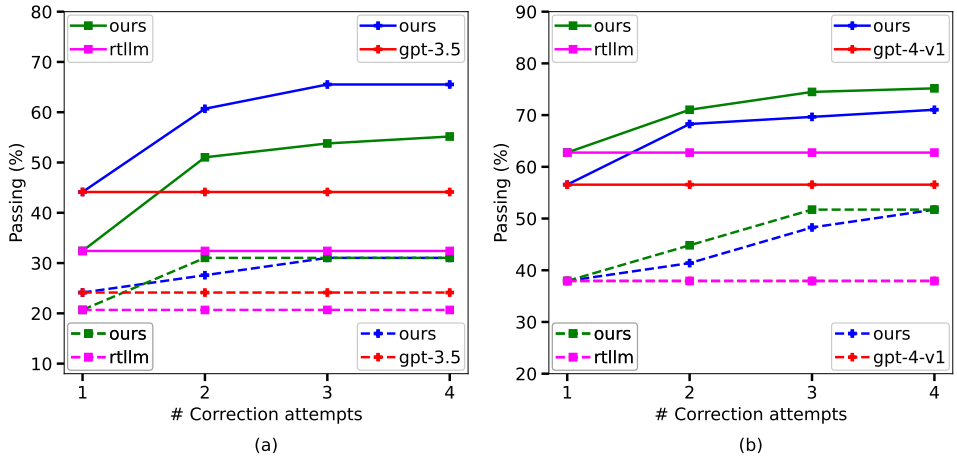

Figure 3: Correctness of generated Verilog code with respect to correction attempt on RTLLM dataset, using (a) GPT-3.5; (b) GPT-4-v1; Dash lines: Syntax; Solid lines: Functionality.

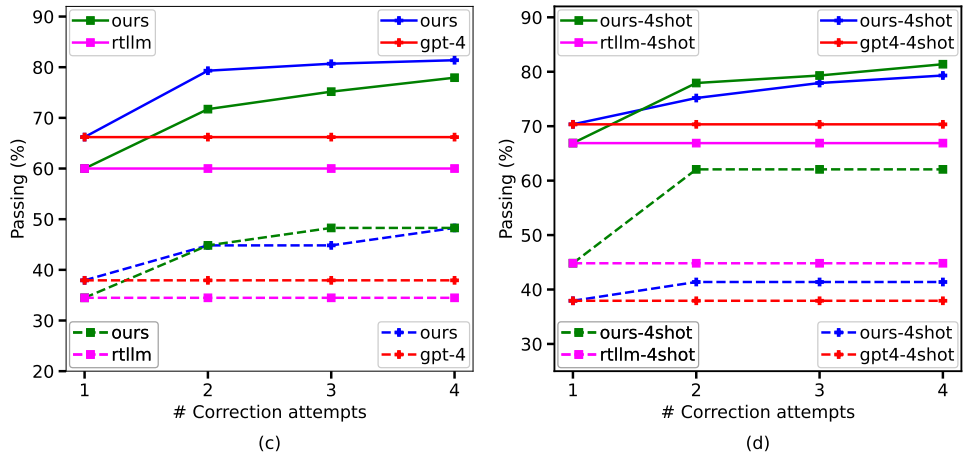

Figure 4: Correctness of generated Verilog code with respect to correction attempt on RTLLM dataset, using (a) GPT-4; (b) GPT-4-4shot. Dash lines: Syntax; Solid lines: Functionality.

Table 1: PPA results of generated Verilog code

Design Name	GPT-4			GPT-4 (4-shot)		
	Clock (ps)	Power (μW)	Area (μm^2)	Clock (ps)	Power (μW)	Area (μm^2)
adder_8bit	318.5	6.3	38.5	333.1	6.1	42.9
adder_16bit	342.2	10.9	104.5	135.1	41.1	152.8
adder_32bit	500.0	14.2	211.6	500.0	14.7	213.2
multi_booth	409.0	112.1	526.0	409.0	112.1	526.0
right_shifter	47.5	144.3	42.9	47.5	144.3	42.9
width_8to16	74.1	223.2	145.8	145.6	128.7	157.2
edge_detect	61.5	49.0	23.3	61.5	49.0	23.3
mux	54.7	215.3	86.1	54.7	215.3	86.1
pe	500.0	552.5	2546.5	500.0	541.0	2488.6
asyn_fifo	295.2	406.4	1279.3	228.3	526.6	1295.4
counter_12	134.4	33.1	40.6	124.5	34.6	36.4
fsm	88.3	32.7	31.5	68.7	49.0	50.2
multi_pipe_4bit	254.7	40.7	131.3	-	-	-
pulse_detect	10.3	187.5	13.5	32.7	59.1	13.5
calendar	-	-	-	208.6	86.6	199.0

4 Evaluation

4.1 Datasets

In assessing our VeriPPA framework, we utilize two benchmark datasets. Firstly, the RTLLM dataset (Lu et al., 2023) includes 29 designs. Secondly, we employ the VerilogEval dataset (Liu et al., 2023), which comprises two subsets: VerilogEval-human, featuring 156 designs, and VerilogEval-machine, consisting of 108 designs.

4.2 Experimental Setup

We demonstrate the effectiveness of our VeriPPA framework for generating PPA-optimized Verilog code for the given designs. We adopt, GPT-3.5 (OpenAI, 2023a) and GPT-4 (OpenAI, 2023b) as our LLM models. We use n=1, temperature temp = 0.7, and a context length of 2048 in our setting. Further, we incorporate the ICARUS Verilog simulator (Williams, 2023) to automate the testing of the generated code. For PPA check, we perform the logic synthesis using Synopsys Design Compiler with `compile_ultra` command and we use the ASAP 7nm Predictive PDK (Vashishtha et al., 2017). We implement an in-house simulator to sweep the timing constraints to find the fastest achievable clock frequency for all the generated designs. All ex-

periments are conducted on a Linux- based host with AMD EPYC 7543 32-Core Processor and an NVIDIA A100-SXM 80 GB.

4.3 Generation Correctness

This study evaluates Verilog code generation accuracy using two primary metrics: syntax checking and functionality verification. Figures 3 and 4, shows the results of our methodology of improving Verilog correctness through successive correction attempts. We generate five different codes for each design description, attempting up to four corrections within each generation. We set correction attempts to four because, after certain attempts, the efficiency of these corrections diminishes, as the LLMs tend to provide repetitive responses to identical errors.

In Figure 3, we plot syntax and functionality correctness percentages against the number of correction attempts. The graph features a solid line for syntax correctness and a dotted line for functionality correctness. Functionality is evaluated the same as RTLLM (Lu et al., 2023), considering a design functionally correct if at least one generated code passes the functionality test. We use GPT-3.5 and observe initial syntax correctness of 44.13% and functionality correctness of 24.13%, as shown in Figure 3 (a). After applying correction attempts, our VeriPPA achieves the correctness of 65.51% for syntax and 31.03% for functionality. Next, we compare to another baseline RTLLM (Lu et al., 2023) which uses a self-learning technique on top of GPT model to improve the correctness, which initially scores 24.82% in syntax and 27.58% in functionality without corrections. The reason for the low accuracy is that the response to self-planning from GPT-3.5 generates different planning approaches but not Verilog codes directly. Please note that integrating our correction approach into RTLLM increases the maximum syntax and functionality correctness to 49.65% and 34.48%, respectively.

We then evaluate two versions of the GPT-4 model. The first, GPT-4-0314 (v1), shows an initial syntax correctness of 56.55% and functionality correctness of 37.93%. Our correction methods could enhance the correctness to 71.03% for syntax and 51.72% for functionality. Our VeriPPA enhances RTLLM's syntax correctness to 75.17% and functionality to 51.72%, from 62.75% and 37.93% respectively, as indicated in Figure 3 (b). For the base GPT-4 model, we notice an increase in syntax correctness from 66.2% to 81.37% by the fourth attempt and in functionality from 37.93% to 48.27%. With our VeriPPA method, RTLLM's syntax correctness further improved from 60% to 77.93%, and functionality from 34.48% to 48.27%, as shown in Figure 4 (a).

Finally, testing the GPT-4 model with four-shot learning, we observe a correctness score improvement in syntax from 70.34% to 79.31% and in functionality from 37.93% to 41.37%. Our method increases RTLLM's syntax correctness from 66.89% to 81.37%. The functionality correctness notably can be increased from 44.82% to 62.06% after four attempts, as demonstrated in Figure 4 (b). This significant improvement highlights the effectiveness of VeriPPA in enhancing the functional accuracy of the hardware designs.

In evaluating our VeriPPA framework with VerilogEval datasets, we found notable improvements. For the VerilogEval-Machine dataset, We show syntax and functionality correctness against the number of correction attempts in Figure 5, our method increases syntax accuracy from 92.11% to 99.56% for GPT-4 case. Functionality correctness also increased from 33.57% to 43.79% using GPT-4, and further to 45.25% with GPT-4's four-shot learning. This shows that the four-shot learning is effective in improving the functionality correctness of the design. The VerilogEval-human dataset shows similar trends, where syntax correctness increases from 91.28% to 97.17% when we use GPT-4. functionality accuracy is improved from 29.48% to 39.74% through the application of GPT-4 and its four-shot learning variant as shown in the Figure 6. This underscores our framework's effectiveness in enhancing both syntax and functionality in Verilog code generation.

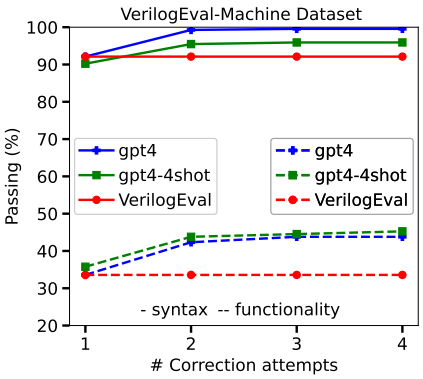

Figure 5: Correctness of generated Verilog code with respect to correction attempt on VerilogEval-Machine dataset. Dash lines: Syntax; Solid lines: Functionality.

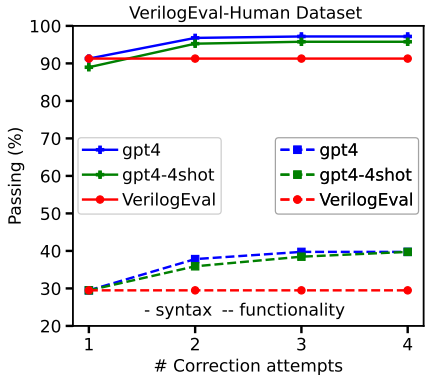

Figure 6: Correctness of generated Verilog code with respect to correction attempt on VerilogEval-Human dataset. Dash lines: Syntax; Solid lines: Functionality.

Figure 7: Optimization Flow, (a) Non-optimized PPA results, (b) PPA constraint-based prompt and ICL, (c) LLM, (d) Optimized results

4.4 PPA Optimization

We use the Synopsis Design Compiler for synthesizing the designs, culminating in the production of PPA reports. The PPA results of complex designs are encapsulated in Table 1. This table, though comprehensive, does not encompass specific design constraints. Similar to the ChipGPT approach (Chang et al., 2023), where an output manager and enumerative search finalize the PPA from multiple reports, our process also generates multiple PPA reports for each design. An example is the $pulse\_detect$ design, which consistently met criteria across five evaluations of passing functionally and syntactically. Therefore, in post-synthesis, we collate five PPA reports for the $pulse\_detect$ design, and we select the most optimized one to include in Table 1.

Table 2: PPA Optimized Verilog Design Results

Design Name	Clock (ps)	Power (μW)	Area (μm)
adder_32bit	180.0	587.31	1005.67
multi_booth	123.2	42.39	42.92
pe	325.0	1206.0	4863.88
asyn_fifo	114.8	988.92	1344.86

It is crucial that PPA results do not conform to specialized design requirements, a standard practice in industrial applications. To address this disparity, we further perform the PPA constraint-based feedback mechanism, integrated with ICL, as illustrated in Figure 7. This approach represents a significant step towards aligning LLM-generated codes with industry-specific PPA requirements. Figure 7 demonstrates our process, starting with the collection of synthesized design outputs that require optimization. For example, $adder\_32bit$, is initially synthesized with a 500ps clock as shown in Figure 7 (a). To enhance the speed of $adder\_32bit$, we impose a clock constraint, aiming for a clock speed of less than 300ps, as outlined in the PPA constraint-based prompt in Figure 7. The framework instructs the LLM to consider various optimization strategies, including Pipelining, Clock Gating, Parallel Operation, and Hierarchical Design. It also encourages the exploration of additional methods to generate Verilog code that meets the defined optimization constraints, as illustrated in the context-based learning segment of Figure 7.

Upon providing the PPA-based constraint prompt and context to the LLM, we analyze the resultant Verilog code for syntax and functional accuracy, making corrections where necessary. If the code passes both checks, we proceed to its final synthesis, achieving an optimized Verilog code as shown in Figure 7 (d), where the $adder\_32bit$ operates at an improved 180ps clock. In Table 2, we present the results of selected optimized designs. Notably, no design from the VerilogEval (Liu et al., 2023) dataset features in Table 2, as those designs did not require complex optimization.

4.5 Line of Codes

We use a scatter plot to illustrate the average number of lines of codes (LOC) for different Verilog files, as shown in Figure 8. The y-axis indicates the percentage LOC and the x-axis denotes the specific circuit categories. It indicates that our generated codes have a wide range of circuit coverage.

4.6 Language construction converage

We further count the occurrences of different modules such as always blocks, module declarations,

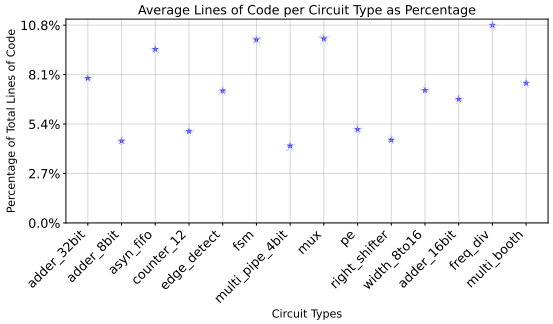

Figure 8: Scatter plot showing the average number of lines of code per circuit type, illustrating the relative complexity and coding effort required for each category.

conditional statements (If statements, else if statements), case statements, and others, these frequencies are then normalized to present their relative percentages in Figure 9. It signifies that our generated codes cover a diverse set of design modules.

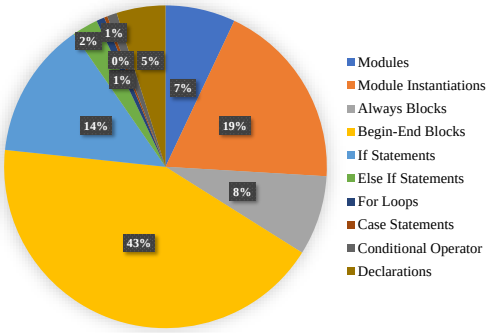

Figure 9: Proportional usage of Verilog constructs, illustrating the dataset's composition and dominant modules.

4.7 t-SNE Visualization

We use the t-distributed Stochastic Neighbor Embedding (t-SNE) for dimensionality reduction and then K-means clustering to group the similar points in the elliptical shapes, as illustrated in Figure 10. The clusters, namely Module-Heavy, Conditional-Intensive, Always-Block-Rich, Timing-Control Dominant, and Complex State Machines, are the top five categories based on frequency. Table 3

Table 3: Distribution of Modules and Clusters

Clusters	Labels	Percentage (%)
Timing-Control Dominant	Begin-End Blocks	42.74
Module-Heavy	Module Instantiations	18.92
	Modules	7.01
	Declarations	4.89
Conditional-Intensive	If Statements	13.70
	Else If Statements	2.61
Always-Block-Rich	Always Blocks	7.99
Complex State Machines	Conditional Operator	0.98
	For Loops	0.82
	Case Statements	0.33

presents the distribution of high-level categories identified through K-means clustering applied to modules in the generated Verilog codes. The percentages reflect the normalized frequency of each module, showcasing the predominant modules and their associated high-level clusters within the Verilog code generation process.

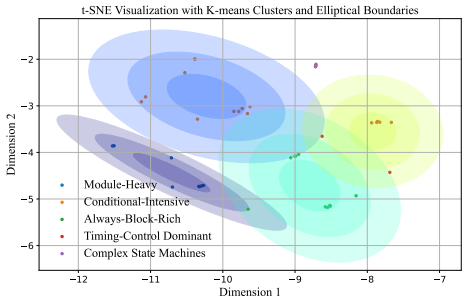

Figure 10: Visualization using t-SNE and K-means clustering, demonstrating the data's segmentation and the underlying structure.

5 Limitations

In this study, we employed GPT-4 for conducting our experiments due to the observed limitations in the accuracy of code generation when using the free available version of LLM, specifically GPT-3.5, which exhibited lower correctness in generated outputs. Consequently, this necessitated the use of GPT-4, leading to increased experimental costs. Moreover, the decision to limit correction attempts to four was informed by empirical evidence indicating that beyond this threshold, the LLM (GPT-4) tends to produce identical Verilog outputs for given prompts with errors most of the time.

6 Conclusion

In this paper, we introduce a novel framework VeriPPA, designed to assess and enhance LLM efficiency in this specialized area. Our method includes generating initial Verilog code using LLMs, followed by a unique two-stage refinement process. The first stage focuses on improving the functional and syntactic integrity of the code, while the second stage aims to optimize the code in line with Power-Performance-Area (PPA) constraints, an essential aspect of effective hardware design. This dual-phase approach of error correction and PPA optimization has led to notable improvements in the quality of LLM-generated Verilog code. Our framework schieves 62.0% (+16%) for functional accuracy and 81.37% (+8.3%) for syntactic correctness in Verilog code generation, compared to SOTAs.

References

Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243*.

Tom Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Kaiyan Chang et al. 2023. Chipgpt: How far are we from natural language hardware design.

Wen Chen et al. 2012. Novel test detection to improve simulation efficiency: A commercial experiment. In *ICCAD'12*, page 101–108, New York, NY, USA.

Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03*, page 286–291, New York, NY, USA.

Winston Haaswijk et al. 2018. Deep learning for logic optimization algorithms. In *2018 ISCAS*, pages 1–4.

HDLBits. 2023. Hdlbits:verilog practice. https://hdlbits.01xz.net. Accessed on 11/20/2023.

Abdelrahman Hosny et al. 2020. Drills: Deep reinforcement learning for logic synthesis. In *2020 25th ASP-DAC*, pages 581–586.

Hanbin Hu et al. 2018. Hfmv: Hybridizing formal methods and machine learning for verification of analog and mixed-signal circuits. In *DAC '18*, New York, NY, USA.

Rongjian Liang et al. 2020. Drc hotspot prediction at sub-10nm process nodes using customized convolutional network. In *ISPD '20*, page 135–142, New York, NY, USA.

Mingjie Liu et al. 2023. VerilogEval: evaluating large language models for verilog code generation. In *ICCAD'23*.

Zeye Liu et al. 2019. Improving test chip design efficiency via machine learning. In *2019 ITC*, pages 1–10.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. Rtllm: An open-source benchmark for design rtl generation with large language model.

Dani Maarouf et al. 2018. Machine-learning based congestion estimation for modern fpgas. In *FPL'18*, pages 427–4277.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023a. Gpt-3.5. https://platform.openai.com/docs/models/gpt-3-5. Accessed on 15/11/2023.

OpenAI. 2023b. Gpt-4. https://platform.openai.com/docs/models/gpt-4. Accessed on 15/11/2023.

Alec Radford et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Shailja Thakur et al. 2023. Benchmarking large language models for automated verilog rtl code generation. In *DATE'23*, pages 1–6. IEEE.

Vinay Vashishtha, Manoj Vangala, and Lawrence T Clark. 2017. Asap7 predictive design kit development and cell design technology co-optimization. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Fanchao Wang et al. 2018. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *GLSVLSI'18*, page 207–212, New York, NY, USA.

Samuel Ward et al. Pade: A high-performance placer with automatic datapath extraction and evaluation through high dimensional data learning. In *DAC'12*, pages 756–761.

S. Williams. 2023. The icarus verilog compilation system. [Online]. Available: https://github.com/steveicarus/iverilog.